Lloyd Fosdick
Guest Editor

Predicative Programming Part II

ERIC C.R. HEHNER

*ABSTRACT: Part I of this two-part paper presented a new semantics of programs. Each program is considered to be a predicate, in a restricted notation, that specifies the observable behavior of a computer executing the program. We considered a variety of notations, including assignment, composition (semicolon), deterministic choice (**if**), nondeterministic choice, definition (nonrecursive and recursive), and variable declaration. We did not consider any input or output notations, or concurrency; that is the subject of Part II. We assume the reader is familiar with Part I, so that we can build on ideas presented there.*

0. SEQUENCE NOTATIONS

We shall describe input and output as sequences of messages communicated on named channels. To do so, we find it convenient to introduce a few sequence notations.

λ	the empty sequence
$\#c$	the length of sequence c
$c[i]$	the element of sequence c with index i, $0 \le i < \#c$ (indices, of course, start at 0),
$c[i\ldots]$	the subsequence of c from index i on, $0 \le i \le \#c$
$c \frown d$	catenation of sequences c and d
$c \ge d$	sequence c is an extension of sequence d, i.e., d is an initial subsequence of c
$c \lesssim d$	sequence c is a tail of sequence d, i.e., c is a final subsequence of d

This research was supported by the National Science and Engineering Research Council of Canada.

In this paper, we shall not be forming sequences of sequences; consequently, we can get away with neglecting to write sequence brackets. For example, we write $c \frown 1$ to mean the sequence formed by extending sequence c with element 1.

1. OBSERVABLES

In Part I, observation of computer activity consisted of observing the values of variables at the start of the activity, and if it terminates, at its end. We now wish to extend our observations to include communications between a computer and its environment during the execution of a program. The environment can be a combination of people and computers. It can even include the same computer that is executing the program in question: a computer can consist of several processors, each capable of executing a program and communicating with the others; a single processor can execute several programs in an interleaved fashion. For this reason, we adopt the standard term "process" to mean the particular computer activity being described.

Input is a communication to a process from its environment along a named channel designated for that purpose. Let c be an input channel. Then $\grave{c}$ denotes the sequence of inputs offered by the environment on channel c during execution. And $\acute{c}$ denotes the sequence of inputs that are offered on c to the process but never consumed by it. The relationship between $\grave{c}$ and $\acute{c}$ is

$$\exists \bar{c}.\ \grave{c} = \bar{c} \frown \acute{c}$$

which says (inputs offered) = (inputs consumed)⌢(inputs unconsumed). More compactly, we write $\acute{c} \lesssim \grave{c}$.

Output is a communication from a process to its environment along a named channel designated for that purpose. Let d be an output channel. Then $\grave{d}$ denotes the sequence of outputs already produced on channel d before the start of the process. And $\acute{d}$ denotes the sequence of outputs produced on d by the end of the process if it terminates, forever if it does not. The relationship between $\grave{d}$ and $\acute{d}$ is

$$\exists \bar{d}.\ \grave{d} \frown \bar{d} = \acute{d}$$

which says (outputs produced before)⌢(outputs produced during) = (outputs produced). More compactly, we write $\acute{d} \geq \grave{d}$.

The vector of values $\grave{v}$ is now extended to include the communication sequences $\grave{c}, \grave{d} \ldots$ for all channels, as well as the initial values of variables. Similarly, the vector of values $\acute{v}$ is now extended to include the sequences $\acute{c}, \acute{d}, \ldots$ as well as the final values of variables. (We continue to pronounce $\grave{v}$ and $\acute{v}$ as "v in" and "v out" although the words "in" and "out" are no longer entirely appropriate.) We postulate that $\grave{v}$ and $\acute{v}$ constitute the possible observations that may be made of a process. This choice can be criticized, but it is convenient and, we hope, reasonable.

Our choice of possible observations means that we can prescribe the order of communications on a particular channel, but not their times; for that, a more elaborate theory would be needed. Our choice also means that we cannot directly prescribe the relative order of communications on different channels. But often, this order will be indirectly constrained by data dependencies. For example, an output that depends on an input must follow it in time.

An input may, in the mind of the user, depend on an output, and so the user will not provide the input until the output is produced. This dependence is not part of the semantics of the process, but part of the semantics of the user. As we shall see, our semantics requires that an output that can be produced without further input must be produced. And so, a (sufficiently patient) interactive user (interactive environment) is accommodated.

A batch environment is one that is willing to provide all input without seeing any output. This environment is also accommodated by our semantics.

2. SPECIFICATIONS

A specification is a predicate whose free variables are $\grave{v}$ and $\acute{v}$. To be achievable, a specification S must obey

(8) $\quad \forall \grave{v}.\ \exists \acute{v}.\ \mathrm{S}$

and be computable (as in Part I); but, now it must also allow the proper relationship between the initial and final communication sequences. We call this relationship K. For example, with input channel c and output channel d,

$$\mathrm{K} = \acute{c} \lesssim \grave{c} \wedge \acute{d} \geq \grave{d}$$

The generalization to any number of channels should be obvious; in particular, if there are no channels, K is *true*.

The specification

(16) $\quad \acute{d} = \grave{d} \frown 0$

is achieved by any process that outputs a 0 (and no more) on channel d. The specification

(17) $\quad \acute{d} = 0 \frown \grave{d}$

satisfies formula (8), but requires a process to change the past. This is obviously not reasonable, so we must strengthen (8); to be achievable, a specification S must obey

(18) $\quad \forall \grave{v}.\ \exists \acute{v}.\ \mathrm{S} \wedge \mathrm{K}$

Thus, (16) is achievable, but (17) is not.

The specification K requires nothing whatever of a process except that it be a process, i.e., that it proceed by consuming inputs and producing outputs, rather than by changing the past. It is the least determined (weakest) process specification, saying only that consumed input cannot become unconsumed, and produced output cannot become unproduced. (The K stands for "chaos.")

3. PROGRAMS

A program is a process specification in a highly restricted notation to ensure that it is automatically achievable. Since every process achieves specification K, every program P must satisfy

(19) $\quad \forall \grave{v}, \acute{v}.\ \mathrm{P} \Rightarrow \mathrm{K}$

Our programming notations for communicating processes include all the notations of Part I, and a few more. Some of the definitions from Part I must now be extended.

3.0 Skip

$$\textbf{skip} =_{\mathrm{df}} \acute{v} = \grave{v}$$

This definition remains the same except that the vectors $\grave{v}$ and $\acute{v}$ are now extended to include communication sequences.

3.1 Assignment

$$x := e =_{\mathrm{df}} ((\mathrm{D}`\grave{e}' \wedge \grave{e} \in \mathrm{type}(x)) \Rightarrow \acute{v} = \grave{v}^{x}_{\grave{e}}) \wedge \mathrm{K}$$

As in Part I, we henceforth neglect the types of variables and assume that expressions are always of the right type for their contexts. Since $(\acute{v} = \grave{v}^{x}_{\grave{e}}) \Rightarrow \mathrm{K}$, we can write the assignment predicate in other forms that are sometimes more convenient.

$$\begin{aligned} x := e = {} & (\mathrm{D}`\grave{e} \Rightarrow \acute{v} = \grave{v}^{x}_{\grave{e}}) \\ & \wedge (\neg \mathrm{D}`\grave{e}' \Rightarrow \mathrm{K}) \\ x := e = {} & \mathrm{D}`\grave{e}' \wedge \acute{v} = \grave{v}^{x}_{\grave{e}} \\ & \vee \neg \mathrm{D}`\grave{e}' \wedge \mathrm{K} \end{aligned}$$

3.2 Input

The notation $c?x$ specifies a process that receives an input on channel c, and assigns it to variable x. To aid our understanding of input, we can consider it equivalent to

$$x := c[0]; \quad c := c[1\ldots]$$

although c is not a sequence variable, but a channel.

$$c?x =_{\text{df}} (\#\grave{c}>0 \Rightarrow \acute{v} = \grave{v}^{\grave{c}}_{\grave{c}[1\ldots]}{}^{\grave{x}}_{\grave{c}[0]}) \wedge \mathrm{K}$$

Again, it may be clearer to separate this into two cases.

$$c?x = \#\grave{c}>0 \wedge \acute{v}=\grave{v}^{\grave{c}}_{\grave{c}[1\ldots]}{}^{\grave{x}}_{\grave{c}[0]}$$

$$\vee\ \#\grave{c}=0 \wedge \mathrm{K}$$

If at least one input is offered on channel c, then one input will be consumed and assigned to x, and all else will be undisturbed. If not, then nothing is specified except that $c?x$ is a process. In an interactive environment, it can never be known that no input is forthcoming, and so execution had better remain consistent with the possibility that an input will eventually be offered. In other words, execution must wait for input. In a batch environment, an error message can be communicated.

3.3 Output

The notation $d!e$ specifies a process that evaluates expression e, and outputs the result on channel d. As an aid, it can be considered equivalent to

$$d := d\frown e$$

$$d!e =_{\text{df}} (\mathrm{D}\text{‘}\grave{e}\text{’} \Rightarrow \acute{v}=\grave{v}^{\grave{d}}_{\grave{d}\frown\grave{e}}) \wedge \mathrm{K}$$

3.4 Composition

In Part I, composition P;Q was defined as

$$(\neg\forall\acute{v}.\ \mathrm{P}) \Rightarrow (\exists\check{v}.\ \mathrm{P}^{\acute{v}}_{\check{v}} \wedge \mathrm{Q}^{\grave{v}}_{\check{v}})$$

The antecedent was true of just those initial values $\grave{v}$ for which we have an interest in some final value. We interpreted this to mean that any mechanism achieving P terminates for those initial values.

If P describes a communicating process, then $(\neg\forall\acute{v}.\ \mathrm{P})$ is true for all initial values $\grave{v}$ due to (19). Even if we are not interested in the result $\acute{v}$, we know $\mathrm{P} \Rightarrow \mathrm{K}$. Composition now becomes

$$\mathrm{P;Q} =_{\text{df}} ((\neg\forall\acute{v}.\ \mathrm{P}=\mathrm{K}) \Rightarrow (\exists\check{v}.\ \mathrm{P}^{\acute{v}}_{\check{v}} \wedge \mathrm{Q}^{\grave{v}}_{\check{v}})) \wedge \mathrm{K}$$

Here are some examples using variables x and y, input channel c, and output channel d.

$c?x;\ c?y$

$$= \quad \#\grave{c}\geq 2 \wedge \acute{x}=\grave{c}[0] \wedge \acute{y}=\grave{c}[1] \wedge \acute{c}=\grave{c}[2\ldots] \wedge \acute{d}=\grave{d}$$

$$\vee\ \#\grave{c}=1 \wedge \acute{c}=\grave{c}[1\ldots] \wedge \acute{d}\geq\grave{d}$$

$$\vee\ \#\grave{c}=0 \wedge \acute{c}=\grave{c} \wedge \acute{d}\geq\grave{d}$$

This can be written more compactly, but for the sake of exposition, we have written it as three alternatives. The first says that if at least two inputs are offered, then exactly two inputs will be consumed and assigned to x and y, and no output will be produced. The second disjunct says that if exactly one input is offered, it will be consumed; nothing is said about the final value of x or y or whether outputs are produced. The last disjunct covers the case that no inputs are offered.

$c?x;\ d!1$

$$= \quad \#\grave{c}\geq 1 \wedge \acute{x}=\grave{c}[0] \wedge \acute{y}=\grave{y} \wedge \acute{c}=\grave{c}[1\ldots] \wedge \acute{d}=\grave{d}\frown 1$$

$$\vee\ \#\grave{c}=0 \wedge \acute{c}=\lambda \wedge \acute{d}\geq\grave{d}$$

$d!1;\ c?x$

$$= \quad \#\grave{c}\geq 1 \wedge \acute{x}=\grave{c}[0] \wedge \acute{y}=\grave{y} \wedge \acute{c}=\grave{c}[1\ldots] \wedge \acute{d}=\grave{d}\frown 1$$

$$\vee\ \#\grave{c}=0 \wedge \acute{c}=\lambda \wedge \acute{d}\geq\grave{d}\frown 1$$

These two examples differ only in one detail: When no input is offered, the first does not require an output but the second does.

$c?x;\ c?y;\ d!x+y$

$$= \quad \#\grave{c}\geq 2 \wedge \acute{x}=\grave{c}[0] \wedge \acute{y}=\grave{c}[1] \wedge \acute{c}=\grave{c}[2\ldots] \wedge \acute{d}=\grave{d}\frown(\grave{c}[0]+\grave{c}[1])$$

$$\vee\ \#\grave{c}<2 \wedge \acute{c}=\lambda \wedge \acute{d}\geq\grave{d}$$

In practice, programming and proving correctness rarely require the calculation of semantics in complete detail. For example, the specification

$$(\#\grave{c}\geq 2 \wedge \#\grave{d}=0) \Rightarrow (\acute{c}=\grave{c}[2\ldots] \wedge \acute{d}=\grave{c}[0]+\grave{c}[1])$$

begins with a typical hypothesis saying that enough inputs will be offered, and no output has yet been produced. To prove that $c?x;\ c?y;\ d!x+y$ is correct, only part of its first disjunct is needed.

3.5 Deterministic Choice

$$\textbf{if } b \textbf{ then } \mathrm{P} \textbf{ else } \mathrm{Q} =_{\text{df}}$$

$$(\mathrm{D}\text{‘}\grave{b}\text{’} \Rightarrow (\grave{b}\wedge\mathrm{P} \vee \neg\grave{b}\wedge\mathrm{Q})) \wedge \mathrm{K}$$

3.6 Nondeterministic Choice

$$\mathrm{P} \textbf{ or } \mathrm{Q} =_{\text{df}} \mathrm{P} \vee \mathrm{Q}$$

3.7 Input Choice

Let a and c be two input channels, let x and y be two variables, and let P and Q be two specifications.

$$[a?x \rightarrow \mathrm{P} \ \Box\ c?y \rightarrow \mathrm{Q}]$$

specifies the following process: If no input is offered on channel a or on channel c, then no result is promised (execution must wait for input). If input is offered on channel a but not on channel c, then one input is consumed from channel a and assigned to x, and P describes the subsequent activity. Symmetrically, if input is offered on channel c but not on channel a, then one input is consumed from channel c and assigned to y, and Q describes the subsequent activity. If input is offered on both channels a and c, then one of them is consumed and the other is not, with the subsequent activity described by the corresponding specification.

The preferred implementation is the one that chooses the first available input, but that can neither be specified nor observed.

$$
\begin{aligned}
&[a?x \rightarrow \mathrm{P} \,\square\, c?y \rightarrow \mathrm{Q}] \\
&=_{\mathrm{df}} \#\grave{a}{=}\#\grave{c}{=}0 \wedge \mathrm{K} \\
&\quad\vee \#\grave{a}{>}0 \wedge P^{\grave{a}\ \ \grave{x}}_{\grave{a}[1\ldots]\,\grave{a}[0]} \\
&\quad\vee \#\grave{c}{>}0 \wedge Q^{\grave{c}\ \ \grave{y}}_{\grave{c}[1\ldots]\,\grave{c}[0]}
\end{aligned}
$$

Input choice can be generalized to any number of alternatives.

THEOREM 13.

(a) $[a?x \rightarrow \mathrm{P} \,\square\, c?y \rightarrow \mathrm{Q}] = [c?y \rightarrow \mathrm{Q} \,\square\, a?x \rightarrow \mathrm{P}]$
(b) $[c?x \rightarrow \mathrm{P} \,\square\, c?x \rightarrow \mathrm{P}] = [c?x \rightarrow \mathrm{P}]$
(c) $[c?x \rightarrow \mathrm{P}] = (c?x;\ \mathrm{P})$

Input choice and nondeterministic choice have some similarities, but also a difference. The two programs

$$[a?x \rightarrow \mathrm{P} \,\square\, c?y \rightarrow \mathrm{Q}]$$

$$[a?x \rightarrow \mathrm{P}]\ \textbf{or}\ [c?y \rightarrow \mathrm{Q}]$$

are identical if input is offered on neither channel, and also if input is offered on both channels. But they differ if input is offered on only one channel; in that case, the input choice describes a process that must consume the input, but the nondeterministic choice describes a process that may consume it or may wait forever for the other input.

3.8 Definition

The (possibly recursive) definition

$$\mathrm{P}\colon \mathrm{F}(\mathrm{P})$$

was given meaning in Part I by forming a monotonically strengthening sequence of predicates

$$\mathrm{P}_0 = \mathit{true}$$

$$\mathrm{P}_{n+1} = \mathrm{F}(\mathrm{P}_n)$$

whose limit was considered to define P.

$$\mathrm{P} =_{\mathrm{df}} \forall n.\ \mathrm{P}_n$$

The first predicate of the sequence P_0 describes, as well as possible, a completely unknown mechanism. With successive members of the sequence, we know more and more about the mechanism, and the sequence limit is considered to be a complete description.

A completely unknown process is described by K, so this is the first predicate P_0. The definition

$$\mathrm{P}\colon \mathrm{F}(\mathrm{P})$$

gives P the meaning

$$\mathrm{P} =_{\mathrm{df}} \forall n.\ \mathrm{F}^n(\mathrm{K})$$

For example, suppose there are no variables and one output channel d.

$$\mathrm{ONES}\colon d!1;\ \mathrm{ONES}$$

The defining sequence is

$$\mathrm{ONES}_0 = \acute{d} \geq \grave{d}$$

$$\mathrm{ONES}_{n+1} = \mathrm{ONES}_{n}{}^{\grave{d}}_{\grave{d}\frown 1}$$

By constructing a few members of this sequence

$$\mathrm{ONES}_1 = \acute{d} \geq \grave{d} \frown 1$$

$$\mathrm{ONES}_2 = \acute{d} \geq \grave{d} \frown 1 \frown 1$$

we are led to propose

$$\mathrm{ONES}_n = \acute{d} \geq \grave{d} \frown \underbrace{1 \frown 1 \frown \ \ldots\ \frown 1}_{n \text{ ones}}$$

which can be written formally as

$$
\begin{aligned}
\mathrm{ONES}_n = {}& \acute{d} \geq \grave{d} \wedge \#\acute{d} \geq \#\grave{d}+n \\
& \wedge (\forall i\colon \#\grave{d} \leq i < \#\grave{d}+n.\ \acute{d}[i]=1)
\end{aligned}
$$

This can be proven by induction on n. Now ONES is

$$
\begin{aligned}
\mathrm{ONES} &= \forall n.\ \mathrm{ONES}_n \\
&= \acute{d} \geq \grave{d} \wedge \#\acute{d}=\infty \wedge (\forall i \geq \#\grave{d}.\ \acute{d}[i]=1)
\end{aligned}
$$

As expected, ONES specifies a process that outputs an infinite sequence of ones. Although we cannot observe an infinite sequence of outputs, we can observe any finite subsequence of it.

Manipulation of infinite sequences is a subject that requires some attention. We do not pursue it here, but we present one more example, in variables x and y and input channels a and c.

$$\text{GO–ON}\colon [c?x \rightarrow \text{GO–ON} \,\square\, a?y \rightarrow \textbf{skip}]$$

$$
\begin{aligned}
&\text{GO–ON} \\
&= (\exists i.\ \#\grave{c}>i \wedge \#\grave{a}>0 \wedge \acute{c}=\grave{c}[i+1\ \ldots] \\
&\qquad \wedge \acute{x}=\grave{c}[i] \wedge \acute{a}=\grave{a}[1\ \ldots] \wedge \acute{y}=\grave{a}[0]) \\
&\quad \vee \#\grave{a}>0 \wedge \acute{c}=\grave{c} \wedge \acute{x}=\grave{x} \wedge \acute{a}=\grave{a}[1\ \ldots] \\
&\qquad \wedge \acute{y}=\grave{a}[0] \\
&\quad \vee \#\grave{c}<\infty \wedge \#\grave{a}=0 \wedge \acute{c}=\acute{a}=\lambda \\
&\quad \vee \#\grave{c}=\infty \wedge (\forall i.\ \acute{c} \lesssim \grave{c}[i\ \ldots]) \wedge \acute{a} \lesssim \grave{a}
\end{aligned}
$$

If input is offered on both channels c and a, then an arbitrary initial portion of c may be consumed, leaving the last consumed input as the value of x, and one input from a will be consumed and assigned to y. According to the preferred implementation of input choice, the amount of c input consumed will depend on the arrival times of the messages, the c input being "interrupted" at some time by an a input. The second disjunct describes the possibility that the "interruption" occurs before any c input has been consumed. In the third disjunct, a finite amount of input is offered on channel c, and none on channel a; in that case, all the input will be consumed, and execution will wait for more (no final values of x and y are promised). And finally, if an infinite sequence of inputs is offered on c, these inputs may be consumed forever.

3.9 Independent Composition

The program

(20) $x := 0;\ y := 1$

specifies a particular desired result: $\acute{x}=0$ and $\acute{y}=1$ and all else unchanged. It can be achieved by a process that first executes $x := 0$ and then executes $y := 1$. The same result can be achieved by a process that first executes $y := 1$ and then $x := 0$. Suppose we have three variables x, y, and z, and two channels c and d. Then

$$\begin{aligned} & x := 0;\ y := 1 \\ & = \acute{x}=0 \wedge \acute{y}=1 \wedge \acute{z}=\grave{z} \wedge \acute{c}=\grave{c} \wedge \acute{d}=\grave{d} \\ & = y := 1;\ x := 0 \end{aligned}$$

A program is a specification relating final values to initial values and describing all mechanisms that achieve the specification. If two processors are available, then program (20) can even be achieved by a mechanism that executes the two assignments concurrently. The same is true of the following three programs

$x := z;\ y := z$

$x := 0;\ d!1$

$c?x;\ d!1$

but not of the programs

$x := 0;\ x := 1$

$x := 0;\ y := x$

$x := y;\ y := 0$

$x := 0;\ d!x$

$c?x;\ d!x$

Composition (the semicolon connective) prescribes sequential execution only to the extent required by data dependencies; beyond that, execution order is left as a freedom for the implementer.

Two programs are called *independent* if

(a) neither contains an assignment or input to a variable appearing in the other, and
(b) no channel appears in both.

The importance of independence is that it is easily recognized by an implementation, and according to the next theorem, it allows P;Q to be executed by executing P and Q concurrently.

If P and Q are independent programs, the variables and channels v can be partitioned into two disjoint and exhaustive groups v_P and v_Q as follows: v_P includes all variables assigned in P, all variables receiving input in P, and all channels appearing in P; v_Q includes all variables assigned in Q, all variables receiving input in Q, and all channels appearing in Q; a variable that is nowhere assigned or used for input may be placed in either group, and similarly for a channel appearing in neither P nor Q. Let $P(v_P)$ be P but restricted to the variables and channels v_P, and similarly $Q(v_Q)$.

THEOREM 14. If P and Q are independent programs,

$$P(v_P) \wedge Q(v_Q) \Rightarrow P;Q$$

If P and Q are independent, any mechanism that achieves $P(v_P)$ and separately also $Q(v_Q)$ achieves P;Q. The implication is, in fact, equality, except only when one of P or Q is completely undetermined.

We introduce a connective for independent programs, called independent composition. P ‖ Q can be achieved by a process achieving P and Q separately.

For independent programs P and Q,

$$P \parallel Q =_{\mathrm{df}} P(v_P) \wedge Q(v_Q)$$

THEOREM 15. If P and Q are independent programs,

(a) $((\neg\forall\acute{v}.\ P=K) \wedge (\neg\forall\acute{v}.\ Q=K)) \Rightarrow (P \parallel Q = P;Q)$
(b) $((\forall\acute{v}.\ P=K) \wedge (\forall\acute{v}.\ Q=K)) \Rightarrow (P \parallel Q = P;Q)$

For example, with variable x and output channel d,

$$\begin{aligned} & \mathrm{ONES} \parallel x := 2 \\ & = \acute{d}\ge\grave{d} \wedge \#\acute{d}=\infty \wedge (\forall i\ge\#\grave{d}.\ \acute{d}[i]=1) \wedge \acute{x}=2 \end{aligned}$$

$$\begin{aligned} & \mathrm{ONES};\ x := 2 \\ & = (\neg\forall\acute{v}.\ \mathrm{ONES} = K) \Rightarrow (\exists\dot{v}.\ \mathrm{ONES}^{\dot{v}}_{\acute{v}} \wedge (x := 2)^{\dot{v}}_{\grave{v}}) \\ & = true \Rightarrow (\exists\dot{x}, \dot{d}.\ \dot{x}=\grave{x} \wedge \dot{d}\ge\grave{d} \wedge \#\dot{d}=\infty \\ & \qquad \wedge (\forall i\ge\#\grave{d}.\ \dot{d}[i]=1) \wedge \acute{x}=2 \wedge \acute{d}=\dot{d}) \\ & = \acute{d}\ge\grave{d} \wedge \#\acute{d}=\infty \wedge (\forall i\ge\#\grave{d}.\ \acute{d}[i]=1) \wedge \acute{x}=2 \end{aligned}$$

The antecedent $(\neg\forall\acute{v}.\ P=K)$ in the formula for (semicolon) composition still indicates "interest" in some "final" value, but it can no longer be interpreted as requiring termination. The program (ONES; $x := 2$) does indeed assign 2 to x.

When one of P or Q is undetermined, (P;Q) is undetermined but (P ‖ Q) is not. Informally speaking, when something goes wrong in P, the composition (P;Q) does not permit us to trust any variable, but the composition (P ‖ Q) permits us to trust v_Q. Similarly, when something goes wrong in Q, (P;Q) does not permit us to trust any variable, but (P ‖ Q) permits us to trust v_P. Independent composition acts as a firewall, limiting damage in case of trouble.

Independent composition is our only connective that is not defined for all predicates. P ‖ Q is defined only when predicates P and Q are independent, and then its meaning depends on the partitioning of v into v_P and v_Q whenever a variable is not assigned or a channel is not used, and one of P or Q is undetermined. In this respect, independent composition is not entirely satisfactory from a mathematical point of view. And, it is the only connective that does not have the property of bounded nondeterminism (Part I, Theorem 6, generalized to include channels). But the engineering concern for partial safety in the face of partial disaster makes it an important connective. It is the connective of distributed computing.

3.10 Variable Declaration

The formula for variable declaration

$$\textbf{var } x.\ \mathrm{P} =_{\mathrm{df}} \exists \acute{x}.\ \mathrm{P}^{\grave{x}}_{\perp}$$

remains unchanged, but its importance is now increased. In Part I, the only observables were the initial and final values of the global variables, so a program having only local variables is of no interest (it is completely undetermined, equivalent to *true*). But a communicating process can be observed through communication, and it is quite reasonable for all variables to be local.

Consider a buffer with input channel c and output channel d. We do not intend it to come after any process producing output on channel d, so we intend to use it when $\grave{d} = \lambda$. We want it to reproduce all that is offered to it on channel c as its output on channel d. Here is a one-place buffer B that introduces a local variable x.

$$\mathrm{B}\text{: } (\textbf{var } x.\ c?x;\ d!x);\ \mathrm{B}$$

Buffer B is correct for the specification

$$(\#\grave{c}{=}\infty \wedge \grave{d}{=}\lambda \Rightarrow \acute{d}{=}\grave{c})$$
$$\wedge\ (\#\grave{c}{<}\infty \wedge \grave{d}{=}\lambda \Rightarrow \acute{d}{\geq}\grave{c})$$

If the amount of input is infinite, it will be reproduced as output. If the amount of input is finite, B promises to reproduce it as output, but not necessarily to stop there, since it is always possible that more input will come. If we want to stop the buffer after a finite amount of input, we must invent a special "STOP" message.

```
B1: var x.
    c?x;
    if x = "STOP"
      then skip
      else (d! x; B1)
```

3.11 Buffered Channel Declaration

Just as variables can be introduced locally within a part of a program, so can channels. Our notation introduces a pair of channels, one for input and one for output, connected so that the output of one is the input of the other.

$$\textbf{chan } c{\leftarrow}d.\ \mathrm{P}$$

A program with channel declarations describes a system of communicating parts.

As with variables, we allow a local channel name to be the same as an existing, global variable or channel name, in which case the local channel obscures the global variable or channel. To express a program with local channels as an assertion about the initial and final values of global variables and channels, the local channels must be renamed if necessary to avoid obscuring anything global.

$$\textbf{chan } c{\leftarrow}d.\ \mathrm{P} =_{\mathrm{df}} \exists \acute{c},\ \acute{d}.\ \mathrm{P}^{\grave{c}\grave{d}}_{\acute{d}\lambda}$$

For $\grave{d}$, we substitute the empty sequence because there can be no previous output on a newly created output channel. For $\grave{c}$, we substitute $\acute{d}$ because the input offered on channel c is exactly the output produced on channel d.

THEOREM 16.
(a) $(\textbf{chan } a{\leftarrow}b.\ \textbf{chan } c{\leftarrow}d.\ \mathrm{P})$
$= (\textbf{chan } c{\leftarrow}d.\ \textbf{chan } a{\leftarrow}b.\ \mathrm{P})$
(b) $(\textbf{chan } a{\leftarrow}b.\ \textbf{var } x.\ \mathrm{P}) = (\textbf{var } x.\ \textbf{chan } a{\leftarrow}b.\ \mathrm{P})$

For our examples, we suppose there are two global variables x and y, and no global channels.

$$(\textbf{chan } c{\leftarrow}d.\ c?x \| d!0) = (\acute{x}{=}0 \wedge \acute{y}{=}\grave{y})$$

In the second example, two inputs are required but only one is produced.

$$(\textbf{chan } c{\leftarrow}d.\ (c?x;\ c?y) \| d!0) = \mathit{true}$$

The result is an undetermined process. (A simpler example is $(\textbf{chan } c{\leftarrow}d.\ c?x) = \mathit{true}$.) In the third example,

$$(\textbf{chan } c{\leftarrow}d.\ c?x \| (d!0;\ d!1)) = (\acute{x}{=}0 \wedge \acute{y}{=}\grave{y})$$

two outputs are produced and only one is consumed. (Similarly $(\textbf{chan } c{\leftarrow}d.\ d!0) = (\acute{x}{=}\grave{x} \wedge \acute{y}{=}\grave{y})$.)

The next example is a deadlocked process.

$$\textbf{chan } a{\leftarrow}b.\ \textbf{chan } c{\leftarrow}d.\ (a?x;\ d!0) \| (c?y;\ b!1)$$
$$= \mathit{true}$$

Each side requires an input before producing the output required by the other side. Consequently, nothing can be said about any final values. If the two sides are willing to produce their outputs before consuming their inputs, no deadlock arises.

$$\textbf{chan } a{\leftarrow}b.\ \textbf{chan } c{\leftarrow}d.\ (d!0;\ a?x) \| (b!1;\ c?y)$$
$$= \acute{x}{=}1 \wedge \acute{y}{=}0$$

This, like the third example, illustrates the fact that the channels we are introducing in this section are automatically buffered.

The notation for channel declaration is easily generalized to introduce one output channel connected to any number (zero or more) of input channels. For example,

$$\textbf{chan } a,\ c{\leftarrow}d.\ \mathrm{P} =_{\mathrm{df}} \exists \acute{a},\ \acute{c},\ \acute{d}.\ \mathrm{P}^{\grave{a}\grave{c}\grave{d}}_{\acute{d}\acute{d}\lambda}$$

introduces three channels a, c, and d such that the output on d is the input on both a and c. Using examples from earlier sections, we have

$$(\textbf{chan } a,\ c{\leftarrow}d.\ \text{GO–ON} \| \text{ONES}) = \mathit{true}$$

Connecting the output from ONES to both inputs of GO–ON allows it to engage in infinite internal chatter (livelock), thus nothing can be said about the final values of x and y. It also allows, but does not require, an input on channel a to interrupt the chatter, so that $\acute{x}{=}\acute{y}{=}1$. But proofs are based on what is assured, not what is possible.

3.12 Synchronous Channel Declaration

In execution terms, a buffered channel from an output to an input means that, although an input command may have to wait until an output is produced, an output command never has to wait. A synchronous channel means that an output command also has to wait until the corresponding input command(s) is(are) ready to be executed.

If we are given only the ability to create buffered channels, as in Section 3.11, we can nonetheless create synchrony as follows. Declare an output-to-input pair d-to-c for message communication, and another pair cc-to-dd for a return acknowledgment, and a variable to receive the acknowledgment.

(21) **chan** $c \leftarrow d$. **chan** $dd \leftarrow cc$. **var** *ack*. P

Within P, output on d should be coded as

(22) $d!e$; dd?*ack*

and input on c should be coded as

(23) $c?x$; $cc!0$

The particular acknowledgment message is irrelevant; only the fact of sending it is significant.

Our notation for declaring a pair of connected channels to provide synchronous communication is

chan $c \Leftarrow d$. P

Semantically, it is equivalent to (21), except that the acknowledgment channels and variable are anonymous, and outputs and inputs within P are to be taken as equivalent to (22) and (23), with the irrelevant acknowledgment message left unspecified.

4. THE FUTURE

We have been careful to incorporate into our semantic formalism the principle that the past cannot be changed. Another principle, which also deserves to be incorporated, is that the future cannot be predicted. Consider the following specification, using variable x, input channel c, and output channel d.

$$\begin{aligned} \text{U: } & \#\grave{c}>0 \wedge \acute{x}=\grave{c}[0] \wedge \acute{c}=\grave{c}[1 \ldots] \wedge \acute{d}=\grave{d} \\ & \vee \#\grave{c}=0 \wedge \acute{x}=1 \wedge \acute{c}=\grave{c} \wedge \acute{d}=\grave{d} \end{aligned}$$

U says to consume an input if there is one and assign it to x, and to assign 1 to x if there is no input; in either case, termination is required. Although U obeys (18), it should be regarded as unachievable: A mechanism can know that there has not yet been any input, but it cannot predict that there will be none and complete its task. No program is correct for specification U.

In addition to the implementation difficulty, there is a logical difficulty when channels are connected. Consider

chan $c \leftarrow d$. U; **if** $x=1$ **then** $d!0$ **else skip**

which has an internal communication if and only if it does not have an internal communication. Our formalism quite properly makes this equivalent to *false*.

We do not yet know the criterion to eliminate prediction. Certainly, for any program P and any input channel c,

$$(\exists \acute{c}.\ \mathrm{P}^{\grave{c}}_{\grave{c}\frown e}) \Rightarrow (\exists \acute{c}.\ \mathrm{P})$$

must be a tautology. This eliminates U, but it is not sufficient to eliminate all dependence on the existence or value of unconsumed input. We leave that to the future.

5. PREVIOUS AND RELATED WORK

This work is a continuation of the effort to express interprocess communication begun by Hoare [2], and further refined in [3] and [1]. The latter work expressed the semantics of communication formally as the strongest describing predicates as does this paper, but the choice of observable variables was different, and it used a single history of communications for all channels. Another paper by Hoare offering a predicate semantics of communicating processes is [4].

A related formalization of communicating processes can be found in the work of Kahn and MacQueen [5]. The ideas concerning the concurrency allowed by semicolon were developed with C. Lengauer [6], who has observed that the lazy-evaluation semantics makes independent composition unnecessary.

Brock and Ackerman [0] have shown that communication histories are inadequate to describe all possible process behaviors. As stated in Part I, it is not our purpose to describe arbitrary mechanisms, but to prescribe desired ones. We believe that communication histories are adequate for the specification of any desired logical behavior of processes. The change from the descriptive to the prescriptive viewpoint is a difficult one, but necessary for an understanding of our choices.

6. CONCLUSION

We have integrated, in one useful semantics, communicating processes (including local channel declaration) and variables with assignment (both local and global, both private and shared). Communication is described by considering channels as sequence variables (queues); output and input are the "join" and "leave" operations. But, unlike ordinary variables, the initial and final values of channel variables must always satisfy a process predicate (K); this means that consumed input cannot become unconsumed, nor produced output become unproduced.

The constructs that we introduced, and their semantics, were not dictated by the formalism. We made a choice, and we invite the reader to experiment with other choices. Finding the best constructs, those that make programming easiest, requires further experience.

Acknowledgments. I thank C. A. R. Hoare, the members of IFIP Working Group 2.3 (Programming Methodology), Christian Lengauer, and Hugh Redelmeier for their comments. Redelmeier provided the paradoxical example in Section 4.

REFERENCES

0. Brock, J.D., and Ackerman, W.B. Scenarios: A model of nondeterminate computation. In: *Formalization of Programming Concepts*, Lecture Notes in Computer Science 107, Springer-Verlag, New York, NY, 1981.
1. Hehner, E.C.R., and Hoare, C.A.R. A more complete model of communicating processes. *Theor. Comput. Sci. 26* (1983), 105–120.
2. Hoare, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (1978), 666–677.
3. Hoare, C.A.R. A calculus of total correctness for communicating processes. *Sci. Comput. Program. 1* (1981), 49–72.
4. Hoare, C.A.R. Specifications, programs, and implementations. Technical Monograph PRG-29, Oxford University Computing Laboratory (Programming Research Group), June 1982.
5. Kahn, G., MacQueen, D.B. Coroutines and networks of parallel processes In: *Information Processing 77.* Proceedings of IFIP Congress 77, Elsevier North-Holland, Inc., New York, 1977, 993–998.
6. Lengauer, C. and Hehner, E.C.R. A methodology for programming with concurrency: An informal presentation. *Sci. Comput. Program. 2* (1982) 1–18.

CR Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies*; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures, input/output*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*logics of programs*
General Terms: Design, Theory, Verification
Additional Key Words and Phrases: predicative programming

Received 11/82; accepted 8/83

Author's Present Address: Eric C. R. Hehner, Computer Systems Research Group, Sandford Fleming Building, University of Toronto, Toronto, Ontario M5S 1A4, Canada.